VIA PadLock
Security Application Note

Version 1.32

This is **Version 1.32** of the VIA PadLock Security Application Note.

LIFE SUPPORT POLICY

VIA processor products are not authorized for use as components in life support or other medical devices or systems (hereinafter called life support devices) unless a specific written agreement pertaining to such intended use is executed between the manufacturer and an officer of VIA.

1. Life support devices are devices which (a) are intended for surgical implant into the body or (b) support or sustain life and whose failure to perform, when properly used in accordance with instructions for use provided in the labeling, can be reasonably expected to result in a significant injury to the user.
2. This policy covers any component of a life support device or system whose failure to perform can cause the failure of the life support device or system, or to affect its safety or effectiveness.

Table of Contents

CHAPTER

1

INTRODUCTION

The recent explosive growth of PC networking and Internet-based commerce has significantly increased the need for good computer and network security mechanisms. VIA is addressing this need with its Padlock Security Suite.

Good security requires good random numbers. Which can be a problem; it is actually hard to generate *good* random numbers on a computer. (For those who do not understand what a random number is, why they are needed in computer applications, what a *good* one is, and how they are generated, Appendix A includes background on these topics.)

To address this need for good random numbers in security applications, VIA introduced the Nehemiah processor in January 2003, which includes a high-performance hardware-based random number generator (RNG) *on the processor die*. PadLock RNG uses random electrical noise on the processor chip to generate highly random values at an extremely fast rate. It provides these numbers directly to the application via a new x86 instruction that has built-in multi-tasking support. By including this unique capability directly within the processor, the VIA Nehemiah processor enables a new level of security for personal computers and embedded devices.

On stepping 8 of the Nehemiah, VIA added a second random number generator on the processor die, which improves the bit generation speed and effective randomness.

Good security also requires good cryptography. Most people understand the need for cryptography; we've all had our secret decoder rings and thrilled to the stories of espionage and the intrigue of spy versus spy.

Good cryptography is hard. Getting the algorithms right, and getting the right algorithms, and using them in a secure fashion (or even in a relatively less vulnerable fashion), is a problem that has plagued the brightest minds for generations. Get it right, and you may or may not get a footnote in the history books and a raise in your next paycheck; get it wrong and your country may lose the war. Or your husband may find out ... (oops, never mind).

Fortunately, there are standard algorithms for performing cryptographic functions. Examples include Rivest-Shamir-Adleman (*RSA*) encryption, the Data Encryption Standard (*DES*) and triple-DES (*3DES*), and more recently, the Advanced Encryption Standard (*AES*), as specified in Federal Information Processing Standards Publication 197 (*FIPS-197*.) These algorithms are symmetric key algorithms, which are characterized by using the same secret key for both encryption and decryption.

Getting an algorithm right is something computer engineers are good at. Making it fast is something we are even better at. Some of our readers may remember calculating trigonometric functions with eight bit integer computers. And how much easier and faster it became when hardware floating point primitives became available on PCs. We can do the same for cryptography.

To address the need for *fast* and *easy-to-use* cryptographic primitives, VIA introduced, on stepping 8 of the Nehemiah processor, a high performance implementation of the Advanced Encryption Standard. This feature, *PadLock Advanced Cryptography Engine (PadLock ACE)*, provides increased cryptographic capabilities for personal computers.

In the VIA Ether processor, the capabilities of PadLock ACE have been enhanced.

There is more to security than generating random bits and a good symmetric key algorithm. There are also public-key algorithms, which use different keys for encryption and decryption. Public-key algorithms can solve subtle problems that occur in protocols using symmetric key algorithms. For example: if we share the same encryption/decryption key, how do we agree on, or exchange, the key when we don't have a secure communications channel? If the channel were secure, we wouldn't need the cryptosystem! Or more commonly, it is the cryptographic algorithms we use that make the communication secure.

Unfortunately, public-key algorithms are much slower than symmetric key algorithms, by many orders of magnitude. Most public-key systems involve modulo exponentiation of very large numbers (integers more than 1000 bits in length).

VIA has addressed the need for faster public-key cryptography by introducing, in the VIA Esther processor, PadLock Montgomery Multiplier (PMM). Dr. Peter Montgomery developed an algorithm some twenty years ago for doing fast modulo multiplication. Since modulo exponentiation is just (a whole lot of!) modulo multiplication and division, the Montgomery algorithm is the basis for fast modulo exponentiation.

There is yet more to security. In addition to keeping communications secure, there is a need for digital authentication. How does Bob know that the carefully encrypted message from Alice really came from Alice and not Charlie?

To deal with this problem, and to create what are known as digital signatures, there are *hash functions*. The National Institute of Standards and Technology (NIST), defines a Digital Signature Standard, or DSS in document FIPS 186-2. A hash function, used in the context of a signature algorithm and in conjunction with an encryption scheme, provides authentication to digital communications.

The VIA Esther processor introduces PadLock Hash Engine (PHE), a fast hardware implementation of the SHA-1 and SHA-256 Secure Hash Algorithms as defined in the Secure Hash Standard, FIPS 180-2.

1.1 RANDOM NUMBER GENERATOR

PadLock Random Number Generator (RNG) has many powerful and unique features.[0] PadLock RNG is available on VIA Nehemiah processors (stepping 3 and higher) and on all VIA Esther processors.

- **Hardware-Based.** Truly random numbers require a non-deterministic source (as opposed to a programmed software algorithm). PadLock RNG bits are obtained from electrical random noise on the processor chip and meet the requirements of truly random numbers: statistically good, unpredictable, and unreproducible.
- **Designed for Verification and Test.** Because PadLock RNG allows direct access to the entropy source, third parties can perform their own tests of the output.
- **Asynchronous Multi-byte Generation.** The hardware generates random bits at its own pace. These accumulate into hardware buffers with no impact to program execution. Software may then read the

[0] Patents pending.

accumulated bits at any time. This asynchronous approach allows the hardware to generate large amounts of random numbers completely overlapped with program execution. This is opposed to good software generators, which can be fast but consume a significant number of cycles.

- **Application-level Access:** Application code can store the random numbers directly from the hardware, without the intervention of privileged device drivers. An x86 instruction is provided to store random data and the associated valid byte count. In addition, a REP version of the instruction allows large amounts of random data to be collected with a single instruction. Direct access ensures that applications receive data directly from the random number generator, avoiding the possibility of receiving nonrandom data due to malicious or accidental device driver flaws.
- **Transparent to the Operating System:** The operations involved in obtaining random bytes and in verifying the configuration all operate in user mode, without operating system knowledge or involvement. For security purposes, changes to the configuration require a higher privilege level.
- **Inherent multitasking (Atomic Instruction):** Storing random data is a single atomic instruction within an application. If a task switch occurs, and a different application wants to store random data, it gets completely new bits.
- **Raw Bit Access.** The raw internally generated random bits may be accessed. Normally the raw bits are filtered through a *whitener* to obtain desirable random characteristics.
- **Controllable Voltage Bias.** The internal voltage bias to the various oscillators can be adjusted. This allows "fine-tuning" of the electrical noise sources and may (or may not) improve the statistical characteristics of the random bits produced.
- **High Bit Rate.** PadLock RNG produces bits at a variable rate. Typical rates vary from approximately 50M to 200M raw bits per second, depending on whether one or two noise sources have been selected. In the default configuration, configured for the "best" randomness, the bit rate will be reduced to between 0.8M and 10M bits per second. Stepping 3 of the VIA Nehemiah processor has one noise source. Stepping 8 of the VIA Nehemiah, and the VIA Esther processor, each have two noise sources. VIA Esther processors may generate random bits at twice the rate of VIA Nehemiah processors, depending on the voltage bias setting.

1.2 ADVANCED CRYPTOGRAPHY ENGINE

PadLock Advanced Cryptography Engine (ACE) has many powerful and unique features.[0] PadLock ACE is supported on stepping 8 of the VIA Nehemiah processor, and on all steppings of the VIA Esther processor.

- **World-Class Data Rate.** Operating on a long series of data blocks, PadLock ACE can encrypt or decrypt data at a sustained rate of 25 Gb/sec.[0], faster than other known commercial AES hardware implementation, and several times faster than software implementations. For a single encryption or decryption instruction operating on 256 data bits, the effective rate can be even faster: up to 40 Gb/sec.[0]
- **Free.** The unique AES implementation is so small (approximately 0.6 mm^2) that its inclusion on VIA processors did not change the die size or any other processor cost factor.
- **Non-Intrusive.** Although PadLock ACE provides a significant amount of function with many options, only one x86 opcode is used. This is the same x86 opcode introduced for the random number generator.

[0] Patents pending.

[0] Asymptotic REP instruction performance, ECB mode, 2 GHz Esther processor.

[0] The timing of the encryption and decryption operations is effectively variable since a large portion of the time can be overlapped with subsequent instruction execution. This rate assumes the maximum possible instruction overlap.

- **Application-level Access.** Application code can directly encrypt and decrypt blocks without the intervention of any privileged code. One non-privileged x86 instruction provides all PadLock ACE functions *without any need for a device driver or for a change to the operating system*. In addition to facilitating PadLock ACE availability, this application-level capability improves security by ensuring that all related cryptographic information is kept within the application.
- **Inherent multitasking.** In addition to any single application being able to directly use PadLock ACE, any number of tasks can use PadLock ACE concurrently without supplemental task management by the application or by the operating system. Although PadLock ACE contains additional processor state information, the using tasks do not need to save and restore this state - the hardware manages the additional state in a transparent fashion.
- **Performs Both Encryption & Decryption.** This seems obvious, but many hardware AES engines support encryption only. In addition to providing both encryption and decryption, PadLock ACE provides them with the same performance.
- **All Three AES Key Sizes Are Supported Directly in Hardware.** FIPS-197 defines three key sizes (128-bits, 196-bits, and 256-bits) but requires only that one key size be supported. Many hardware AES engines support only the basic 128-bit key size. PadLock ACE directly supports all three key sizes in hardware with the same performance.
- **Five Operating Modes Supported Directly in Hardware.** The most common encryption *operating modes* are implemented in hardware. These modes are *Electronic CodeBook* (ECB), *Cipher Block Chaining* (CBC), 128-bit *Cipher FeedBack* (CFB), *Counter* (CTR), and 128-bit *Output FeedBack* (OFB).[0] Providing these operating modes directly in hardware improves performance (versus implementing them in software) and improves the integrity and security of the encrypted data. Most other AES devices support only one or two of these modes in hardware. **Note**: CTR mode is supported only on the VIA Esther processor.
- **Multi-block pipelined.** For ECB operating mode, PadLock ACE hardware is pipelined such that it concurrently operates on two data blocks at the same time, thus doubling the effective data rate.
- **High Performance REP Function.** The basic encryption and decryption instructions all use the x86 REP prefix, to allow large amounts of data to be encrypted or decrypted with only one instruction. Unlike x86 string operations, where multiple string operations are (slightly) faster than REP forms, the REP encryption and decryption instructions are much faster than a series of comparable single encryption instructions - generally 50% or more.
- **Intermediate Round Results Provided.** PadLock ACE implements a special mode that returns the result of any particular round of the multi-round AES algorithm. Properly calculating this intermediate result is not obvious since the algorithm for intermediate rounds is different from the last round of the normal algorithm.
- **User Defined Round Count.** The user specifies the round count used in the encryption or decryption. Normally, this will be the round count defined in the AES standard, but some users may prefer a custom round count for research, improved performance, or increased security.
- **Two Extended Key Generation Option**s. For 128-bit keys, PadLock ACE can directly perform the AES-defined expansion of the user-supplied key into the extended keys used by the AES rounds. Alternately, the application may directly provide a standard or non-standard extended key sequence directly to the hardware. This may be useful for cryptographic research or using a private encryption algorithm.
- **Message Authentication**. PadLock ACE has been extended in the VIA Esther processor to support message authentication, adding CBC-MAC and CFB-MAC modes to the existing CBC and CFB modes.

[0] As defined in *NIST Special Publication 800-38A, Recommendations for Block Cipher Modes of Operation*, 2001

1.3 PADLOCK HASH ENGINE

PadLock Hash Engine (PHE) has many powerful features[0]. PHE is available on all steppings of the VIA Esther processor.

- **Free.** The unique PHE implementation is so small (less than 0.5 mm2) that its inclusion on VIA processors did not change the die size or any other processor cost factor.
- **Non-Intrusive.** Although PHE provides a significant amount of function with many options, only one x86 opcode is used. This is the same x86 opcode introduced for PadLock Montgomery Multiplier.
- **Application-level Access.** Application code can directly hash messages without the intervention of any privileged code. One non-privileged x86 instruction provides all PHE functions *without any need for a device driver or for a change to the operating system.* In addition to facilitating PHE availability, this application-level capability improves security by ensuring that all related cryptographic information is kept within the application.
- **Support for both SHA-1 and SHA-256**. Both of the most commonly used Secure Hash Algorithms are directly supported by the hardware.
- **Inherent multitasking.** In addition to any single application being able to directly use PHE, any number of tasks can use PHE concurrently without supplemental task management by the application or by the operating system. Although PHE contains additional processor state information, the using tasks do not need to save and restore this state - the hardware manages the additional state in a transparent fashion.

1.4 PADLOCK MONTGOMERY MULTIPLIER

PadLock Montgomery Multiplier (PMM) has many powerful and unique features[0]. PMM is available on all steppings of the VIA Esther processor.

- **Free.** The unique PMM implementation is so small (less than 0.5 mm2) that its inclusion on VIA processors did not change the die size or any other processor cost factor.
- **Non-Intrusive.** PMM uses the same x86 opcode introduced for PadLock Hash Engine.
- **Application-level Access.** Application code can use PMM without the intervention of any privileged code. A non-privileged x86 instruction executes PMM *without any need for a device driver or for a change to the operating system.*
- **Inherent multitasking.** In addition to any single application being able to directly use PMM, any number of tasks can use PMM concurrently without supplemental task management by the application or by the operating system. Although PMM contains additional processor state information, the using tasks do not need to save and restore this state - the hardware manages the additional state in a transparent fashion.
- **Variable modulus:** Most hardware-assist big-integer units are very restrictive in the sizes of the big integer operations that they support. A typical unit may only support one or two sizes, typically 1024 or 2048 bits. PMM supports big integer modulo multiplication in the range from 256 bits to 32768 bits, inclusive, in increments of 128 bits.
- **Compatible data structures:** The data structures used for large integers by PMM are compatible with those used by **GMP**, the **G**NU **M**ultiple **P**recision Arithmetic Library.

[0] Patents pending

[0] Patents pending

CHAPTER

2

RANDOM NUMBER GENERATOR

2.1 TEST METHODOLOGY

The statistical test suites described in Appendix A were developed to test software random number generators. These generators are necessarily deterministic, and provide only as much entropy as found in the initial seed - *regardless of the number of "random bits" generated*. While Centaur Technology has made extensive use of these tests, we have found that simpler tests that check for a biased statistical distribution are more appropriate and more effective for testing our hardware generator.

During the initial development of PadLock RNG, while we were getting the bugs out and trying to understand the characteristics of the part, we tested the device with the NIST suite, the DIEHARD suite, and many others. These tests proved valuable in helping us to understand some of our earlier failures, and in comparing PadLock RNG with other hardware random number devices and software algorithms.

However, once we had stable hardware, we determined that a simple chi-square test of the distribution of the 256 possible byte values was more sensitive to deviations from "randomness" - *for PadLock RNG* - than any other test. That is to say that the NIST tests, and the DIEHARD tests, were frequently passing (the P-values were within the confidence levels at the expected frequency) when the chi-square test showed failure.

We continue to use the above test suites (particularly the DIEHARD tests) on a regular basis and have incorporated them into our comprehensive test harness. But they are no longer part of the routine testing.

(Remember that many of the tests in the above test suites are designed to catch flaws in deterministic random generators that were already known to have good statistical distributions.)

As noted in the introduction, the raw bits delivered by PadLock RNG display a small bit-to-bit correlation. The (software selectable) hardware one-of-N filter reduces this correlation to small levels, such that statistical tests performed on samples of (say) 10MB of random bits will pass the above statistical tests, allowing for the fact that - as Marsaglia says - "P happens". A failure is to be expected, occasionally, if you perform enough tests.

However, even with the hardware one-of-N filter set at the maximum value, the measured entropy of the bits from PadLock RNG is still slightly less than 1. That means that a suitably designed test, on a *sufficiently large* sample of random bits, will *consistently* fail.

Once our standard methodology was in place, extensive testing of production steppings of PadLock RNG was performed. Centaur Technology has sampled and tested in excess of *50 terabytes* of data produced by PadLock RNG. The tests were (*and continue to be*) run on many different chips running on several different PC platforms running several different operating systems, but most tests were performed using Red Hat Linux 7.1 (kernel version 2.4.2-2)

Many thousands of samples, usually of 10 MB each, were taken from each chip/system, using all eight DC bias settings. The samples were taken with the hardware one-of-N filter set to the maximum value, to obtain the best possible results. When stepping 8 of the Nehemiah processor became available, we performed the tests on each noise source separately and with both noise sources active.

Many thousands of samples, also of 10MB each, were taken with the hardware one-of-N filter off, and with the whitener disabled. While the statistical properties of these samples of raw bits was poor, that is to say they consistently failed the statistical tests, these samples allowed us to measure the entropy of the raw bits produced by PadLock RNG.

A smaller set of samples was taken at intermediate settings, with the whitener enabled but with the one-of-N filter set to less than the maximum value. Most of these samples were taken early in our development process, which enabled us to determine the appropriate maximum value for the one-of-N filter.

For every sample, the standard chi-square test on the byte distribution was performed. The cumulative distribution of the delivered bits was also subjected to the chi-square test, and a Kolmogorov-Smirnov goodness-of-fit (KS) test on the distribution of the chi-square values for the individual samples was performed. We also measured the effective generated bit rate. And on some samples we performed DIEHARD, NIST, or other statistical tests.

As noted above, the entropy of PadLock RNG is not a perfect 1 bit of entropy per bit, but somewhat less. Thus, we have seen that for very large samples of bits, the chi-square test for the entire sample, or the KS test of the distribution of the P values from the chi-square tests of the individual 10MB samples (or both), will eventually fail. *Even though each 10MB sample passes our statistical tests*.

Now, *when* the cumulative distribution chi-square test or the KS test will fail varies with the DC bias setting, from part to part, and slightly on other environmental conditions such as temperature or what the experimenter had for breakfast that morning.

Typically, we set an arbitrary limit on the P-value for the cumulative chi-square statistic, and on the KS statistic. Then, when either test exceeded that limit (sometimes we required that both tests have P values outside our confidence level), testing at that bias setting was terminated, a different bias value was set, and the tests restarted. Of course, the reports of the test harness were saved to disk. Generally, however, the random bits themselves were not saved, except when we intend to evaluate them later with the NIST or DIEHARD test suites.

(Hey, do you have 50TB of disk drives lying around? We can fill a 30 GB drive with raw bits in a little more than 20 minutes - *if* the drive controller can keep up with us.)

This procedure would repeat, continuously, for a week or two, when a different chip would be placed in the system and the entire process began anew. This can be thought of as trying to estimate a "Mean Time Before (Statistical) Failure" for PadLock RNG. Evaluation of these data allowed us to select, as the default, that DC bias setting that gave the best results, on average.

Most tests were run at ambient conditions. A few were run and analyzed at the worst-case temperature conditions and at intermediate temperatures. The bottom line is that temperature is just one of the variables

that contribute to the entropy of PadLock RNG. Furthermore, the effect of changes in temperature on the bit generation rate varies with the DC bias setting.

As an interesting comparison, and to validate our test harness, the same statistical tests were also performed on random numbers from other sources. One source is the standard Linux software random number generator **/dev/urandom** running on the same PC platforms. This generator gathers environmental noise from device drivers and other sources into an entropy pool. From this entropy pool random numbers are created using the SHA-1 hash.

A second alternate source is random data provided by www.random.org. This data is derived on atmospheric noise sampled as electrical noise (static) on a radio receiver.

2.2 RESULTS AND RECOMMENDATIONS

An independent evaluation of VIA PadLock Random Number Generator has been performed by Cryptography Research, Inc., of San Francisco.

A copy of the report is available at: www.cryptography.com/resources/whitepapers/VIA_rng.pdf

From their closing commentary:

> *Our analysis indicates that the Nehemiah core random number generator is a suitable source of entropy for use in cryptographic applications. The RNG can be easily incorporated within existing software applications and operating systems and functions well within a multi-application environment. The device meets the overall design objective of providing applications with a high-performance, high-quality, and easy-to-use random number generator.*

NOTE: we have *no idea* how this compares with other hardware generators. VIA PadLock RNG is so much faster than other generators that we haven't been able to obtain enough large samples to test. **We do not consider pre-generated samples suitable for testing.**

Cryptography Research (if you have *any* security or cryptographic requirements for an RNG, that report is recommended reading) measured the per-bit entropy of the raw bits from PadLock RNG, and found that the value lies between 0.78-0.99 bits of entropy per raw bit. These numbers correspond closely with entropy measures taken during our internal testing. We generally measured numbers at the higher end of that range; Cryptography Research was creative in setting up a test environment that produced the lower measured entropy. (Again: go read that report. I'll wait.)

Thus, depending on the paranoia required of your application, you may wish to mix 20, 24, 28, or even 32 bytes (or more!) of raw bits to generate 20 random bytes.

During our testing we frequently mixed raw bits with SHA-1, mixing from 20 to 30 bytes of raw bits to produce 20 bytes worth of random bits. Because of the asynchronous nature of PadLock RNG, the delivered bit rate is only slightly sensitive to the number of input bytes when mixed with a software implementation of SHA-1. We recommend using 24 raw bytes as input to the hash algorithm when using this technique: the multiple of 8 maximizes the bit rate from the RNG, and the 24 byte to 20 byte protects against the per-bit entropy being as low as 0.8333 bits per bit.

With a 1.2Ghz Nehemiah processor (stepping 8), we have measured sustained data rates of over 50M bits/second. With a 1.0Ghz Esther processor, using the SHA-1 feature of PadLock Hash Engine instead of a software implementation, we have measured data rates in excess of 170M bits/second. *Bits mixed with SHA-1 have been indistinguishable from random on all tests, including our chi-square and KS tests on samples as large as a ten terabytes*

If you are sufficiently paranoid, increase the number of raw bytes hashed from 24 to 32. This protects against per-bit entropy as low as 0.625 bits per bit. On that same 1Ghz Esther processor discussed above, with the bit rate set to a maximum and using PadLock Hash Engine, the date rate still exceeded 150M bits/second.

Provided that the above entropy measures of VIA PadLock RNG are accurate, and that there are no cryptographic flaws involved in mixing bits with SHA-1, these bits should be considered cryptographically secure, or truly random.

In addition to the Linux **/dev/urandom** device discussed above, there is also a **/dev/random** generator on Linux platforms that functions the same as **/dev/urandom**, except that the estimated entropy of the output random numbers does not exceed the estimated entropy of the underlying entropy pool. Thus these bits are considered cryptographically secure. The bit rate for **/dev/random** with VIA Nehemiah (stepping 8) processors with the 2.4.2-2 kernels is on the order of a few 100 bits/second.

Many operating systems have incorporated PadLock RNG into their entropy pools. These include Linux kernels (version 2.6 and later), OpenBSD (version 3.3 and later), and FreeBSD. For customers with these operating system kernels, the advantages of PadLock RNG are already part of your system.

Centaur may be able to provide advice or assistance to customers with unusual requirements based on our experience. Please refer to our PadLock Random Number Generator Programming Guide for usage details and contact information.

CHAPTER

3

ADVANCED CRYPTOGRAPHY ENGINE

3.1 ADVANCED ENCRYPTION STANDARD

PadLock Advanced Cryptography Engine (ACE) implements the Advanced Encryption Standard (AES) algorithm. The AES is a modern symmetric-key encryption algorithm intended to replace the older Digital Encryption Standard, and has been selected by the National Institute of Standards and Technology (NIST) as the approved symmetric-key algorithm for United States government use. Use of AES has been approved by the National Security Agency (NSA) for SECRET and TOP SECRET communications.

AES was selected after a multi-year evaluation and selection process involving many – if not most – of the world's finest cryptographers. Many algorithms were submitted. The eventual winner of the process was an algorithm known as **Rjindael**, after it's developers Joan Daemen and Vincent Rijmen, from the Netherlands.

AES encrypts and decrypts data blocks of 128 bits, using cipher keys with lengths of 128, 192, and 256 bits. The original Rjindael algorithm supported additional block sizes and key lengths, but the AES standard did not adopt (and PadLock ACE does not support) other block sizes or key lengths.

AES, like most symmetric key algorithms, mixes the bits in each data block with bits from the cipher key through a number of separate **rounds** in the encryption engine. For some algorithms, including AES, the cipher key is altered for each round, so that although each round in the engine may perform the same operations, the bits from the key differ for each round, which makes cryptographic attacks against the algorithm more difficult. The AES standard defines a specific number of rounds for each of the three defined key sizes:

AES-128	10 rounds
AES-192	12 rounds
AES-256	14 rounds

Why is it that the number of rounds differs for each key size, and why is more than one round necessary?

Symmetric-key algorithms are designed not only with security in mind, but also with an eye to speed. The primitive operations in each round are typically chosen to be very fast on the current (and foreseeable) crop of computers, particularly personal computers. Operations that are fast are usually simple. Since the

algorithm is known, and since an attacker can frequently specify the plaintext to be encrypted with some unknown key, algorithms using only one or two rounds can be quickly broken. This is called a “chosen plaintext attack”. (Unlike modern physics with its quarks and color forces cryptographic naming conventions are bland).

Cryptographers spend a great deal of time studying each symmetric-key algorithm to determine the minimum number of rounds necessary for that algorithm to be secure against known cryptographic attacks. Then they add more rounds to allow for the faster computers of the future. As a “fudge factor” they add still more rounds that they hope will be enough to secure the algorithm against more powerful but yet unknown attacks.

The number of rounds specified by the FIPS-197 standard is thought by cryptographers to be more than sufficient. However, PadLock ACE supports all three key sizes at up to 15 rounds as an additional safety factor. Note that for almost all uses of the AES with PadLock ACE you will be required to use the FIPS 197 specified number of rounds.

3.2 MODES OF OPERATION

The basic AES algorithm operates on each 16-byte date block separately. Used in this basic way, identical plaintext data blocks will produce identical ciphertext blocks when encrypted with the same key. In some scenarios, this creates cryptographic weaknesses.

This basic AES mode, known as Electronic Code Book (ECB) is supported directly by PadLock ACE hardware engine. Because each block is encrypted separately, PadLock ACE takes advantage of the pipelined characteristics of the hardware and encrypts (or decrypts) two blocks at once when used in ECB mode.

To overcome these (sometimes) security weaknesses, cryptographers have developed a number of additional modes for using the AES algorithm. In each of these modes, these is some form of feedback so that the first data block (sometimes as plaintext, sometimes as ciphertext, depending on the mode) is mixed with the second data block, the second data block with the third, and so on.

In addition to ECB mode, PadLock ACE supports these additional AES modes of operation:

- **Cipher Block Chaining** (**CBC**). In this mode, the result of each encryption is XOR'd with the next incoming data block. This cipher block chaining (forwarding) operation insures that any single block of cipher text cannot be decrypted to plain text; only the entire file can be decrypted. This forwarding means that only one block is encrypted at a time; no pipelining of the round engine occurs. On the first round, an Initialization Vector (IV) provided by the application is used in place of the forwarded cipher text.
- **Cipher Feed Back** (**CFB**). In this mode, the Initialization Vector (IV) is first encrypted and the result is XOR’d with the plaintext (to produce ciphertext) or with the ciphertext (to produce plaintext). This XOR result is then encrypted as the IV for the next round. Like CBC mode, only the entire file can be decrypted.
- **Output Feed Back** (**OFB**). In this mode, the Initialization Vector is encrypted and the result is forwarded to the next round for further encryption, yielding a sequence of 128-bit blocks. Think of it as a "one-time pad". This sequence is XOR'd with the plaintext to produce the ciphertext, or with the ciphertext to reconstruct the plaintext. It is thus critical that no 128-bit block of the encrypted sequence be re-used. Again, only the entire file can be decrypted, not just selected blocks.
- **Counter Mode** (**CTR**) Counter mode is very much like OFB mode, in that the Initialization Vector is encrypted and the result XOR’d with the plaintext. It can also be used to pre-calculate a byte stream as a pseudo “one-time pad”. However, instead of the encrypted IV being forwarded to the

subsequent round, the original value of the IV is incremented by one, and this new "counter" is encrypted as the IV for the subsequent round. However, the entire 128-bit Initialization Vector is not treated as a 16-byte integer and incremented. When using Counter Mode, a portion of the IV is selected as a **nonce** and a never-to-be-reused random value is placed in those bits. A good use for PadLock RNG! The remaining bits of the IV, usually 16 bits or 32 bits, are incremented as an unsigned integer field (without overflow into the nonce). VIA PadLock ACE implements CTR mode using the last two bytes of the 16-byte counter as a 16 bit big-endian integer. Reasons for this specific integer field size and format may be found in the VIA Advanced Cryptography Engine Programming Guide. CTR mode is supported on the VIA Esther processor only.

3.3 TEST AND VALIDATION

Validating a cryptographic algorithm is - compared with testing a random number generator - almost trivially easy. After all, it's just an algorithm, with a large number of publicly available data sets with known input and result values. And unlike random tests, where a small number of test failures is to be expected, a cryptographic engine is either correct or it is not. One test failure is one too many.

Centaur Technology has developed a comprehensive test suite, involving both encryption and decryption, at all three key sizes, in all supported operating modes, with varying numbers of input blocks, using both methods of key generation, and with special tests for every corner case we have been able to think of. We have tested a number of processors in a variety of multi-tasking environments, with as many as 10 tasks concurrently running versions of our ACE test package.

No ACE failures have been observed in production steppings of the processor. But don't take our word for it. Cryptography Research, Inc. has completed a (brief) security evaluation of the Advanced Cryptography Engine. A copy of their report is available on the VIA web site:

http://www.via.com.tw/en/images/Products/eden/pdf/CRI%20Evaluation%20of%20PadLock%20ACE.pdf

It is also posted at Cryptography Research:

http://www.cryptography.com/resources/whitepapers/Centaur_AESstatement.pdf

Customers interested in testing ACE for themselves should obtain a copy of the VIA PadLock Advanced Cryptography Engine Programming Guide for programming details. They may also request (please refer to the Programmers Guide for contact information) a copy of our test package.

Many Operating Systems, including OpenBSD, FreeBSD, and some of the Linux kernel distributions have already included support for Padlock ACE. Check with your OS vendor (or read your source code!) to determine whether or not Padlock ACE is directly supported.

CHAPTER

4

HASH ENGINE

4.1 SECURE HASH ALGORITHM

Federal Information Processing Standards Publication (FIPS) 180-2 defines the Secure Hash Standard. As part of the standard, FIPS 180-2 specifies four secure hash algorithms. These algorithms are used to provide a condensed representation, a **message digest**, of electronic data - an electronic message or perhaps a file stored on your hard drive. They are called secure because:

> ... it is computationally infeasible, 1) to find a message that corresponds to a given message digest, or 2) to find two different messages that produce the same message digest.[0]

The idea is that this guarantees a message's integrity. There are times when it is not particularly important that a message be a secret. In fact, it is often the case that you want a message to be in the clear so that everyone can read it. *"I, Tim Tightwad, declare that I am not and will not be responsible for the debts of Tina Spendthrift."* You just want to make sure that it wasn't tampered with in transmission and that there is some way of authenticating that you were the one who sent the message. Tim might be annoyed if Tina could delete both occurrences of the word "not" while the message was in transit to the newspaper, and make it appear the he nevertheless sent the message.

This ability to authenticate a message creates methods of creating digital signatures. There are many digital signature algorithms. NIST has produced publication FIPS 186-2, which specifies the Digital Signature Standard (DSS), and the Digital Signature Algorithm (DSA). Key components of DSA are the hash algorithms defined by FIPS 180-2.

There are many uses for a hash function besides full-fledged digital signatures. Many operating system kernels make use of a hash known as MD-5 for straightforward computing reasons. (Hash tables provide an efficient method of representing and storing certain sparse data structures).

As hash functions are widely used in computing and in security, and as providing fast cryptographic/security primitives is the goal of Centaur and VIA with the PadLock security suite, the VIA Esther processor incorporates a hardware implementation of the Secure Hash Algorithm (SHA) known as PadLock Hash Engine (PHE).

[0] FIPS 180-2 page 1

PHE supports both the original SHA-1 algorithm as defined by FIPS 180-1, and the newer SHA-256 algorithm specified in FIPS 180-2. The two additional hash algorithms defined by FIPS 180-2, SHA-384 and SHA-512, are not implemented in this version of Padlock Hash Engine (PHE).

4.2 HASHES AND RANDOM NUMBERS

In addition to normal uses for a cryptographic hash function, using SHA-1 as a mixing function improves the statistical properties of the output from random number generators. (See Chapter 2) So much so, that with the VIA Esther processor, Centaur recommends setting PadLock RNG whitener OFF and the EDX filter to zero to obtain the maximum bit rate from PadLock RNG, and then mixing the raw bits with one of the Secure Hash Algorithms supported by PHE.

Of course, you can mix bits from PadLock RNG with a software implementation of SHA-1. In fact, that is what most operating system kernels do with the bits they collect into their entropy pools. (Well, they mix with *some* hash function. SHA-1 is perhaps the most common). But using PHE is much faster.

How many raw bits to use for the input byte stream to the hash depends on how paranoid you are. Please see chapter 2 of this application note, the VIA PadLock Random Number Generator Programming Guide, or any of a number of security textbooks for guidance in setting your paranoia level.

CHAPTER

5

MONTGOMERY MULTIPLIER

5.1 PUBLIC KEY ENCRYPTION

In symmetric-key encryption, for example the AES algorithm supported by PadLock ACE, the same cipher key is used to encrypt and decrypt a message. Since both the sender and the recipient of an encrypted message must have the same key, and since the key is the only secret thing about the algorithm, symmetric-key algorithms are sometimes known as secret-key algorithms.

Symmetric-key algorithms are very fast, and with sufficiently long cipher keys (such as those mandated by FIPS-197) are thought to be secure against cryptographic attacks. What this means is that there are no known methods for decrypting a message without knowledge of the cipher key that are significantly faster than a brute-force attack. (A brute force attack tries every possible key and looks for output that looks like "real text". The nature of symmetric key algorithms is such that you will recognize the plaintext when you see it.)

There are cryptosystems known **as one-time pads** that are provably secure, even against brute force attacks, assuming that the cipher key (the one-time pad) is a secret. Why doesn't everyone just use one-time pads? Well, because the secret one-time pad must be just as long as your message. And you can never use the same one-time pad twice, or the provably secure system is trivially insecure. (For an interesting historical anecdote, look up the Venona intercepts on Google)

Actually, you could use PadLock RNG to fill two CD/RW with random bits, keep one for yourself and give one to your partner. If you use them carefully, and nobody makes an extra copy of that CD/RW, you have a one-time pad cryptosystem. But you need a different pair of CD/RW for each person with whom you communicate. Realistically, the chance of user error with these one-time pads is greater than the risk of using a symmetric-key or public-key algorithm. But since you can generate the random bits to fill a 600MB CD/RW in less than a minute with PadLock RNG you do have that choice.

A better way to handle the problem of key exchange is with a public-key cryptosystem. A public-key system is characterized by having different (but related) keys for encryption and decryption. The trick is that with knowledge of one key the ability to figure out the other key is **computationally infeasible**. That's a technical term for "we know how to do it but it takes a god-awful long time".

There are even protocols using public-key algorithms that allow two persons unknown to each other to create a secret cipher key for use with a symmetric key algorithm, even when they don't trust each other. And with the right protocol, this key cannot be determined by anyone who is observing all of your communications.

For example, when you order merchandise from a secure web site, a public-key protocol has been executed down in the innards of your networking stack.

Unfortunately, public-key systems are much too slow to be used for communication of most messages. We're talking orders of magnitude slower than symmetric-key systems.

To support faster public-key cryptosystems, VIA has introduced PadLock Montgomery Multiplier (PMM) in the VIA Esther processor.

5.2 WHAT IS A MONTGOMERY MULTIPLIER?

The most important function in almost all public-key cryptosystems (RSA, for example) is exponentiation modulo some modulus **M** where **M** is a very large integer, typically of length N = 1024 or N = 2048 bits (or more). That is, one must calculate

$$\mathbf{A}^{\mathbf{B}} \bmod \mathbf{M}$$

where **A** and **B** are large integers (necessarily smaller than **M**) also of length N.

Even ignoring the problem of reducing the result modulo **M**, this takes a long time. There are many difficult and elegant methods for reducing the number of primitive arithmetic operations required (32 bit x 32 bit multiplies, 64 bit + 64 bit adds, with a whole lot of shifting going on)

However, so long as we work with our normal, everyday integers (well, actually, a 1024-bit integer is a decimal number of more than 300 digits: the number of particles in the known universe is a number smaller than 100 digits in length. So, unless you are a cryptographer, your everyday experience does not include numbers this big!), we have to reduce each multiply operation by performing a division, as **A * A** is rather likely to be larger than **M**, and modulus is the remainder after the division (**A*A) / M**.

The normal integers, mod **M,** form what is known mathematically as a *field*. Mathematicians have discovered (or invented – it takes a dissertation-length discussion to explain that difference) many other fields. One of the properties of a field is that there is an operation in the field conventionally called multiplication, which is more or less comparable to ordinary multiplication.

In some cases one can transform two ordinary integers into numbers in one of these alternate fields. Then, one performs the "multiply" operation within that field, and then transforms the "product" back to ordinary integers. The result will be exactly what you would have calculated with an ordinary multiply.

This is also true of "division" in a field, since division is just multiplication by the inverse, and another property of a field is that an inverse exists for every number, except for "zero".

There is a particular collection of fields known to mathematicians as the Galois fields $GF(2^K)$. Trust me; you don't need to know more. An interesting property of these fields is that the division operation to reduce a number modulus **M** is just a binary right shift.

So it is *much* faster to perform modulo multiply of two large numbers in this field because you can replace a very slow division operation with a very fast right shift. It is even possible to construct hardware so that as you calculate the partial products of the multiply of **A * B**, the reduction modulo **M** (the right-shift) can be performed in parallel.

This is what we at Centaur Technology have done.

Many existing software implementations of $\mathbf{A}^{\mathbf{B}}$ mod **M**, in particular when used as part of a public-key cryptosystem, already make use of the Montgomery algorithm. However, there is one drawback to using the Montgomery algorithm, which is that it is computationally expensive to calculate the "magic numbers" that are used to transform the regular integers into the Galois field. But this calculation only needs to be done once for each modulus **M**, so that when you need to calculate $\mathbf{A}^{\mathbf{B}}$ mod **M**, which involves many modulo multiplies, the time gained by using the Montgomery algorithm much more than compensates for the time lost in getting started.

APPENDIX

A

ON RANDOM NUMBERS

For those who do not already know what random numbers are, what applications use them, how they are generated, and why most current generators do not really produce random numbers, this appendix provides a general background.

Apology to mathematicians, statisticians, security experts, and information scientists: this background introduction intentionally simplifies some very complex technical topics and skips many important considerations.

A.1 THE NEED FOR RANDOM NUMBERS

To help understand the definition of a random number for computer applications, we need to understand something about how they are used. Historically, many computer applications have required random numbers as an essential component of their logic. For example,

Monte Carlo simulations of physical phenomena in physics, chemistry, mechanical engineering, electrical engineering, meteorology, statistics, aerodynamics, hydrodynamics, and other similar disciplines.

Numeric analysis solutions for many mathematical problems such as solving integral and differential equations, solving linear and non-linear equations, etc.

Casino games and online gambling (to simulate card shuffling, dice rolling, etc.)

Creation of lottery numbers

Generating data for statistical analysis

Generating data for psychological testing

Computer games (choosing random behaviors, dealing cards, etc.)

Computer music generation

These applications have different requirements for their random numbers. For example, computer games generally require neither high performance in generating random numbers nor particularly high degrees of randomness. Other applications, such as psychological testing, require higher degrees of randomness, but do not require high performance generation. Large-scale Monte Carlo-based simulations, however, use millions, billions, or more random numbers, and thus have very high performance requirements.[0] These applications also require good statistical properties of the random numbers, but unpredictability is not particularly important. Other applications, such as online gambling, have very stringent randomness requirements as well as stringent non-predictability requirements (the next number cannot be easily predicted from the previous numbers).

While these historical applications are important to PCs, today the major need for quality random numbers is *computer security*. The recent explosive growth of PC networking and Internet based commerce has significantly increased the need for a wide variety of computer security mechanisms. High quality random numbers are essential to all of the following major components of computer security:[0]

Confidentiality. Data encryption is the primary mechanism for providing confidentiality. Many different encryption algorithms exist (symmetric, public-key, one-time pad, etc.), but all share the critical characteristic that the encryption/decryption key *must not* be easily predictable. The cryptographic strength of an encryption system depends on the strength of the key: that is, the difficulty of predicting, guessing, or calculating the key. To ensure that keys cannot be easily guessed, random number generators are used to produce cryptographic keys and other secret parameters in virtually all serious security applications. Many public key encryption protocols, such as RSA encryption using PKCS #1, also require reliable sources of randomness.

Many successful attacks against cryptographic algorithms have focused not on the encryption algorithm but instead on its source of random numbers. As a well-known example, an early version of Netscape's Secure Sockets Layer (SSL) collected data from the system clock and process ID table to create a seed for a software pseudo-random number generator. The resulting “random” number was used to create a symmetric

[0] James E. Gentle, *Random Number Generation and Monte Carlo Methods*, Springer, 2002

[0] This document assumes some knowledge of security concepts, functions, and protocols. If you need some technical background on these areas and more about random numbers, we recommend the following three classics:

Bruce Schneier, *Applied Cryptography: Protocols, Algorithms, and Source Code in C*, John Wiley & Sons, 1996.

Alfred J. Menezes et al., *Handbook of Applied Cryptography*, CRC Press, 1997

Donald E. Knuth, *The Art of Computer Programming: Volume 2, Seminumerical Algorithms*, AddisonWesley, 1998. Chapter 3 covers random number theory in some detail.

key for encrypting session data. Two graduate students broke this mechanism by developing a procedure for accurately guessing the random number in less than a minute and thus guessing the session key.[0]

Authentication. Challenge/response authentication protocols require that challenge values be unpredictable to ensure that attackers cannot re-use data from previous authentication transactions. The strength of passwords used to protect access and information also depends on the difficulty of predicting or guessing the password. As a result, strong random number generators are necessary to automatically generate strong passwords.

Integrity. Digital signatures and message digests are used to guarantee the integrity of communications over a network. Random numbers are often used in digital signature algorithms to make it difficult for a malicious party to forge the signature. Many signing algorithms, including the U.S. Government's Digital Signature Standard[0] also require random sources to ensure the security of the signing keys.

In summary, good security requires good random numbers.

A.2 WHAT IS A RANDOM NUMBER

The notion of a random number is a non-intuitive concept. What is randomness? Can any *single* number be considered random? Can any *finite sequence* of numbers be considered random? Is it the *numbers* themselves or the *process* that generated the numbers that defines randomness? These questions are the subject of many technical papers, doctoral theses, and bar fights among philosophers, mathematicians, statisticians, and information scientists.

One essential fact emerges from this work: ***there is no such thing as a random number***: there are only methods to produce random numbers.[0] A number by itself (or any finite sequence of numbers) is not random, or non-random. (Is zero any less random than 42?) A practical definition of randomness must focus on the characteristics of the random number generator as well as the characteristics of numbers it generates.

To identify the desired characteristics, we consider the theoretical definition of a perfect random number generator. A *truly random number* generator is one where the probability of each generated bit being a zero (or one) is exactly ½, regardless of what it has generated before.[0] For example, an unbiased coin flip should generate truly random numbers. This definition has many implications beyond the statistical characteristics of the bits generated. It implies that the generator is *unpredictable* (one can't predict the next coin toss better than ½ the time, and *unreproducible* (two separate coin flipping experiments give independent results).

Entropy is a related term that we will use a lot in this document. In general, entropy is the measurement of randomness or chaos. Something that has high entropy is chaotic and unpredictable. Something that is ordered has low entropy. In cryptography, entropy is usually measured in bits. A truly random one-bit number has an entropy of one bit. Entropy is directly connected to the information content of a message. For example, a sequence of *n* random bits (i.e., a sequence having an entropy of *n* bits) cannot on average be expressed or defined in less than *n* bits.

The problem with the theoretical definition of randomness is that the amount of entropy cannot be measured perfectly. It can be easy to demonstrate that a particular generator is *not* random based on its design (for example, its design causes the same sequence of generated numbers to recur ever so often). Unfortunately, demonstrating randomness is much more difficult. Indeed, it is impossible to *prove* that the output from a given random number generator is truly random. As a result, the best we can do is to analyze the statistical characteristics of the previous generated bits and compare them to a mathematically perfect generator.

---

[0] http://www.demailly.com/~dl/netscapesec/wsj.txt

[0] Digital Signature Standard (DSS)", Federal Information Processing Standard Publication 186, May 19, 1994.

[0] John von Neumann

[0] There are many different definitions used in the literature, but they are all equivalent to this definition.

For security applications, the most important characteristics of random numbers are (paraphrasing Schneier[0]):

Unbiased Statistical Distribution. Truly random numbers have certain statistical characteristics. These characteristics can be calculated for a theoretical random generator and compared to the statistics for a generated sample. Based on this comparison, the probability that the sample came from a biased source can be calculated.

For example, one important statistical characteristic is the number of 0 or 1 bits in the sample. For a sample of 20,000 bits from a truly random generator, the probability is 0.0001 that the number of 1 bits is less than 9,725 or more than 10,725. Thus, if we analyze 10,000 samples of 20,000 bits each, and find many samples that deviate from these limits, we should suspect that our generator is not truly random (we should expect only one deviation out of 10,000 tests from a truly random generator).

Of course, this test does not prove randomness. For example, if our generator is stuck at generating alternate 1s and 0s, it would pass this monobit test even though the output is clearly non-random. Thus many different statistical characteristics must be analyzed in order to feel comfortable about a generator. Our alternating 1s and 0s sequence would quickly fail many other tests such as a bit-to-bit correlation test, or a multi-bit block frequency test.

Security applications minimally require good statistical characteristics of the generated random numbers. A generator that meets only this non-biased requirement is called *pseudo-random*.

Unpredictability. In addition to good statistical properties, truly random numbers must be unpredictable; that is, the probability of correctly guessing the next bit of a sequence of bits should be exactly ½ regardless of the values of the previous bits. Some applications do not care about this unpredictability characteristic, but it is critical to security applications.

An example of a predictable sequence that has good statistical characteristics would be a generator that produces, for example, 20,000 statistically good random bits and then repeats the exact sequence again. To identify this problem would require analyzing the samples larger than 20,000 bits. A worse problem with predictability is the use of typical software generators. For any software generator to be unpredictable requires that the software algorithm or its initial values be *hidden*. (If we know the algorithm and initial values, we can produce the same sequence as the software generator.) From a security viewpoint, a *hidden algorithm* or *magic number* approach is very weak if it is possible for someone to predict the output. For example, the previous section mentioned a well-known case where the security of a predictable (software) random number generator failed because the starting values could be guessed easily.

A generator which deterministically derives its output from a starting value, but which meets both the statistical and unpredictability requirement is called *cryptographically secure pseudo-random*. (To be precise, these RNGs meet the unpredictability requirement by making it *computationally infeasible* to predict output bits. An adversary with unlimited computational power could guess possible values for the starting state until one is found that matches the output.)

While some statistical tests may indicate a degree of predictability, whether a generator is considered unpredictable is primarily based on it's inherent design.

Irreproducibility. To meet this requirement, two of the same generators (or the same generator at two different times), given the same starting conditions, must produce different outputs. Clearly, software algorithms do not meet this requirement; only a hardware generator based on random physical processes can generate values that meet this stringent security requirement. There is no test to determine irreproducibility; this determination is based on the inherent design of the generator.

A generator that meets all three requirements is called *really*, or *truly, random*.

---

[0] Schneier (1996) p.45-46

A.3 HOW TO GENERATE RANDOM NUMBERS

We now know what random numbers are and why they are critical to many applications, especially security. Why do we need anything special, however, to generate them? Isn't there a random number generator included in every programming library? Well, there is, but as Schneier points out: "these [software] random number generators are almost definitely not secure enough for cryptography, and probably not even very random. Most of them are embarrassingly bad."[0] This section explores this problem and describes the three basic approaches to generating random numbers: software, physical sources, and quantum randomness, and some combinations of these techniques.

The most common approach to generating random numbers is using a deterministic algorithm implemented by a computer program. Such deterministic algorithms cannot generate truly random numbers. (At best they are predictable and reproducible, and at worst, have bad statistical characteristics.) The quality of software based random number generators is often summarized with John Von Neumann's famous quote: "Anyone who considers arithmetic methods of producing random digits is, of course, in a state of sin."[0] Thus, software generators are usually called *pseudo-random* or *quasi-random* generators. There is, however, a wide range of quality among software generators with some being considerably better for security purposes than others.

Many different software algorithms are used, but all comprise two basic steps for each value generated:

- **Choose a starting value or *seed*.** All software generators start with some initial value, or seed. Some set of arithmetic operations (the mixing algorithm) is performed on this initial seed to produce the first random result. In most cases, this result is then used as the seed to produce the second result, and so forth. In some cases, a new seed is constructed for each value based on external (to the random number generator) events (for example, the time-of-day clock).
- ***Mix* the bits from the seed in an attempt to increase its *apparent* entropy.** The mixing algorithms can be grouped into two distinct groups. Conventional generators such as typically provided for the C language `rand()` function perform simple arithmetic operations. Newer algorithms designed specifically for cryptography perform much more complicated (and time-consuming) operations. Each algorithm has different performance characteristics (bit-generation rate), and different randomness characteristics. Although the mixing operation does not technically increase the total entropy (deterministic process cannot create entropy), it is believed that well-designed mixing operations can make it *computationally infeasible* to distinguish the output from a true random source.

Let's look at the conventional approach, which represents most extent generators. These algorithms take the previous value generated and they apply some arithmetic operation to it to produce the next value. Historically, little theory was applied to the mixing algorithm: if it seemed to mix up the seed bits a lot, it was considered a good algorithm. As an example, Knuth references an *extremely* complicated algorithm he invented in 1959 that, when executed, quickly converged to a constant value. This led him to conclude: "random numbers should not [be] generated with a method chosen at random. Some theory should be used."[0] Subsequently, a significant amount of theory has been applied to defining and analyzing good mixing algorithms. The result is a set of *linear* mixing algorithms that are relatively fast and have reasonably statistical characteristics.

A basic example of a linear algorithm is widely used in modern software generators. It is the *linear congruential method*. The calculation is simple:

$$x_{n+1} = (ax_n + c) \bmod m$$

[0] Schneier (1996) p44

[0] John Von Neumann (1951). Quoted in Knuth, 1968, Vol. 2

[0] Knuth, pg 6

where the constants *a*, *c*, and *m* meet some very specific mathematical constraints and the calculations are performed in the word size of the processor. If the constants are chosen correctly, then the cycle length of the generator is *m*. That is, after *m* numbers, the generated sequence repeats itself. This is, for example, the approach used in the Microsoft Visual C++ 5.0 library implementation of `rand()` where $m = 2^{31}$ (0x80000000).

There are several other linear algorithms that have good theoretical underpinnings. These theoretical characteristics are not intuitively obvious, however. Inventing your own algorithm or making minor changes to a known good algorithm can yield very bad randomness.

Although the cycle length of the Microsoft `rand()` example is too short for Monte Carlo simulations and other applications requiring large amounts of random, linear algorithms *can* have a reasonably long cycle. Similarly, well-designed linear algorithms *can* generate numbers with reasonable statistical characteristics. In general, however, the most common software algorithms do not have statistical characteristics as good as truly random values.

Even worse, these algorithms are easily predictable; that is, knowing the algorithm and the constant values (which are often fixed in the software), one can predict the entire sequence. Even if the seed value isn't constant, it can be guessed or, knowing the last generated value (or any other previously-generated value), one can trivially predict the sequence. These generators are also reversible: given the algorithm and a value, previous values can be easily calculated. These cyclical and predictable characteristics are a disaster for security applications.

The good news is that extensive research, publication, and analysis has been done for years on software random number algorithms for use in security applications. One result is the design of some improved non-linear software generators that closely match the statistical characteristics of a truly random generator and are somewhat unpredictable. These are called *cryptographically secure pseudorandom bit generators* (CSPRBG). They use large internal calculation sizes (typically 64- to 1024-bits) and use cryptographically strong *one-way* algorithms. They use values that are determined dynamically rather than built into the code.

Strong one-way functions make it is impracticable to calculate subsequent or previous values given the current value unless the generator's state is known. Even if an adversary discovers the state at a particular point, the algorithms are not reversible, making it infeasible to determine the number prior to the compromise. Of course, knowing the state of the RNG does allow future results to be calculated. As a result, these algorithms rely on secrecy of the initial conditions and computational intermediates to ensure unpredictability.

A simple example of such a CSPRBG is the Blum-Blum-Shub generator.[0] Here is an abstract:

> Start by selecting two large primes, *p* and *q*, with certain characteristics and calculate the hidden value $n=pq$. Choose some seed, x_0, with certain characteristics relative to *n*.
> Calculate each new random value by
>
> $$x_{i+1} = x_i^2 \bmod n$$
> $$z_{i+1} = \textit{the least significant bit of } x_{i+1}$$

Even if *n* is known, its particular numeric characteristics make the difficulty of calculating the previous value very high.

As good as these CSPRBGs are compared to conventional linear software algorithms, however, they still have some problems:

- They are not widely available.
- They are ~~also~~ prone to typical software mistakes in the implementation.
- They are often computationally very slow.

[0] (In choosing an example, I couldn't resist this name.) See Menezes et. al, pg. 186

- They are predictable in the sense described above (i.e., forward prediction if the initial conditions are known).
- Like all software algorithms, they are reproducible: they produce the same results starting with the same input

Their security relies on an assumption that adversaries do not have enough computational power to perform certain mathematical operations. As a result, their irreversibility cannot be proved, although, in practice, this is not a major problem.

There is a way to improve the situation here. Rather than relying solely on a software generator to produce all of the randomness, a software generator can be used in conjunction with a source of true randomness (entropy). Instead of using the previous result as the seed for the next number to be calculated, output from the independent source of entropy can be used as the seed. Even if the entropy source is imperfect (provides less than one bit of entropy per output bit), the results in this case can be more robust than what could be generated by either the software or the entropy source alone.

An example is using a hardware random number generator (see the next section) to produce the seed for a software algorithm. Although a variety of designs are used, the software algorithms are often based on cryptographic *hash functions.* A hash algorithm mixes up its input bits in a fashion such that its output bits should be unpredictable unless the entire input is known. Thus, a good hash algorithm effectively destroys any statistical correlations among the input bits and makes it computationally infeasible to recover the input from the output. The most commonly used cryptographic hash algorithm, the *Secure Hash Algorithm (SHA-1),*[0] is a U.S. government standard and has been extensively analyzed.

The primary reasons for using a hybrid approach (as opposed to a pure hardware-based or pure entropy based approach) are (1) the historically slow speed of entropy and hardware generators, and (2) the lack of perfect randomness in the hardware or entropy generators. While the VIA Nehemiah processor RNG provides both high performance and high quality output, users may choose to use it in conjunction with software based algorithms.

In terms of randomness characteristics, *entropy generators* based on physical phenomena fall in between software generators and quantum based hardware generators. The idea here is to sample some **"***random***"** physical process and assume the samples are truly random. Unfortunately, computers and software tend to be very predictable, so designers must be very careful to ensure that they collect good entropy.

For example, a widely used PC encryption program derives its keys from several seconds of mouse movements and keystroke timing (directed by the program). The Linux operating system has random number generators (/dev/random and /dev/urandom) that use entropy generated by the keyboard, mouse, interrupts, and disk drive behavior as the seed. Microsoft's CryptGenRandom function (part of the Microsoft CryptoAPI) is similar. It uses, for instance, mouse or keyboard timing input, that are then added to both the stored seed and various system data and user data. Examples of such are the process ID and thread ID, the system clock, the system time, the system counter, memory status, free disk clusters, the hashed user environment block, as the seed. Many other similar environmental sources of randomness have been tried.

While these physical activities may look random, their randomness cannot be proven, and they run the risk of generating poor entropy (or no entropy) if the sampled physical activity is dormant or repetitive. There are several potential security vulnerabilities when using such physical activities. For example, in networked applications such as browsers, the application traffic between a client and server effectively publishes the locations and sequence of the client's mouse-events. Similarly, users may enable "snap-to" options that center the mouse pointer in the center of the button to be pressed and make the click locations predictable. As a result, the entropy from mouse movements in these environments could be far less than an RNG designer expected. Similarly, asking the user to create entropy using the keyboard creates bias since humans tend to follow certain patterns in typing (such as a tendency to use the center of the keyboard).

---

[0] FIPS 180-1, Secure Hash Standards, U.S. Department of Commerce/NIST, 1995

In summary, it is dangerous to use the entropy values *directly* as random numbers. Instead, the entropy values are usually used as the seed to a good hash algorithm to produce the final random numbers. (In the Microsoft CryptoAPI and Linux examples, a SHA-1 algorithm is applied to the entropy seeds). Other common problems with entropy generators on computers are that they require hooks in the operating system, they are difficult to test, they often require some user involvement, and they are slow (since they are based on macro physical events).

An unusual example of a physically based random source is available through www.random.org, which continually provides generated random numbers using atmospheric noise (sampled as radio frequency noise). The rate varies, but this generator typically produces about 40,000 bits per second.

By now you should be convinced that good security requires good random numbers, and while it is possible to generate pretty good statistically random numbers using software, it is difficult, slow, and subject to numerous security pitfalls. The only truly random generator is some mechanism that detects quantum behavior at the sub-atomic level. This is because randomness is inherent in the behavior of sub-atomic particles (remember quantum mechanics, Heisenberg's Uncertainty Theorem, etc.).

A quantum based hardware generator is actually practical. Examples that have been used are:

- The interval between the emission of particles during radioactive decay. This technique is used at a web site to generate random bits that are available on-line from the decay of Krypton-85.[0] This source generates only 30 bytes per second and requires a cumbersome (and dangerous?) pile of hardware.
- The thermal noise across a semiconductor diode or resistor. This is the approach most often used in add on PC hardware.
- The frequency instabilities of multiple free-running oscillators. This approach is the basis of VIA PadLock RNG. While implemented differently than the resistor based approach, ultimately, the source of randomness is the same.
- The charge developed on a capacitor during a particular time period.

These sources have been used in a few commercially available add on random number generator devices, none of which have achieved much visibility or use. Since they are peripheral devices such as PCI cards and serial port devices, these commercial hardware generators are expensive and cumbersome. In addition, they are made by universities, or small companies, with limited marketing and distribution capability. Some examples of such add-on hardware generators are found in these references.[0]

There are two notable exception to the add-on strategy. One is, of course, PadLock RNG. And Intel provided a hardware random number generator in the 82802 Firmware Hub.[0]

A.4 HOW TO VERIFY RANDOMNESS

Given the differences in RNG algorithms and the inherent theoretical problems with software generators, we have the obvious question: how do we compare or quantify random number generator *goodness*? For any serious application or generator design, it is essential to have tests that *quantify* randomness.

---

[0] http://www.fourmilab.ch/hotbits/

[0] http://faust.irb.hr/~stipy/random/hg324.html
http://comscire.com/index.htm
http://mywebpages.comcast.net/orb/
http://www.protego.se/sg100_en.htm

[0] Benjamin Jun et al., *The Intel Random Number Generator*, Cryptography Research, Inc., 1999, available from http://www.cryptography.com/resources/whitepapers/IntelRNG.pdf

In one sense, the definition of a random number is simple: the probability that each bit generated is a one (or zero) should be ½. Unfortunately, that definition doesn't help us directly since we can't actually test the *probability* for the next bit to be generated. All we can do is to study bits that are actually generated and analyze their characteristics to see how likely it is that the generator is random based on what it has generated. This approach turns out to be complicated, and in fact, there is no single or simple set of tests that can accurately identify randomness.

The underlying difficulty here is that truly random numbers occur randomly. That is, every possible value can be generated by a truly random source. If we sample 64 bits, for example, from a truly random source, there is a tiny but non zero probability that all 64 bits will be zero. In that case, should we reject the source as not random? A potential answer is to collect many 64 bit samples and check to see how many times all zeros appears. Statistically, we should expect one all zero set per 2^{64} samples. It is usually impractical to collect such a comprehensive set of samples, however. Indeed, because all outputs should be equally unlikely, no particular output provides *proof* of non-randomness. As a result, even the appearance of multiple all zero sets would not prove that the source is nonrandom. Conversely, what if the sample values are 0, 1, 2, 3, 4, etc.? The values and bits seem to be appearing in the right statistical frequency, but do we really believe that the source is random?

Solutions to this fundamental problem are complex and rely on some understanding of statistical concepts. This testing topic is very important, however, and requires some attention in order to determine the proper use of any RNG, including VIA PadLock RNG. Please see Chapter 2 for a discussion of VIA’s test methodology.